CST2550 Software Engineering Management and Development Coursework 2 – Karaoke Machine – Final Report

Table of Contents

1 Summary....2
2 Introduction....2
3 Design....3
3.1 Search library for a song....4
3.2 Add a song to the library by name....4
3.3 Play the first song in the playlist (and therefore remove it from the playlist)....5
3.4 Add a song to the playlist....6
3.5 View the contents of the playlist....6
3.6 Delete any song in the playlist....7
3.7 Pause the song video playback....7
3.8 Skip to the next song....8
4. Testing....9
4.1.1 ImportMultipleSongs....9
4.1.2 addSong....9
4.1.3 searchLibrary....10
4.1.4 addToPlaylist....10
4.1.5 viewPlaylist....10
4.1.6 deleteFromPlaylist....11
4.1.7 playSong....11
4.1.8 pauseSong....11
4.1.9 skipSong....12
4.1.10 convertCurrentSongtoMediaPlayer....12
5. Conclusion....12
5.1 Summary....12
5.2 Limitations....14
5.3 Future Approach Changes....14
6. References....15
7. Appendices....15

1 Summary

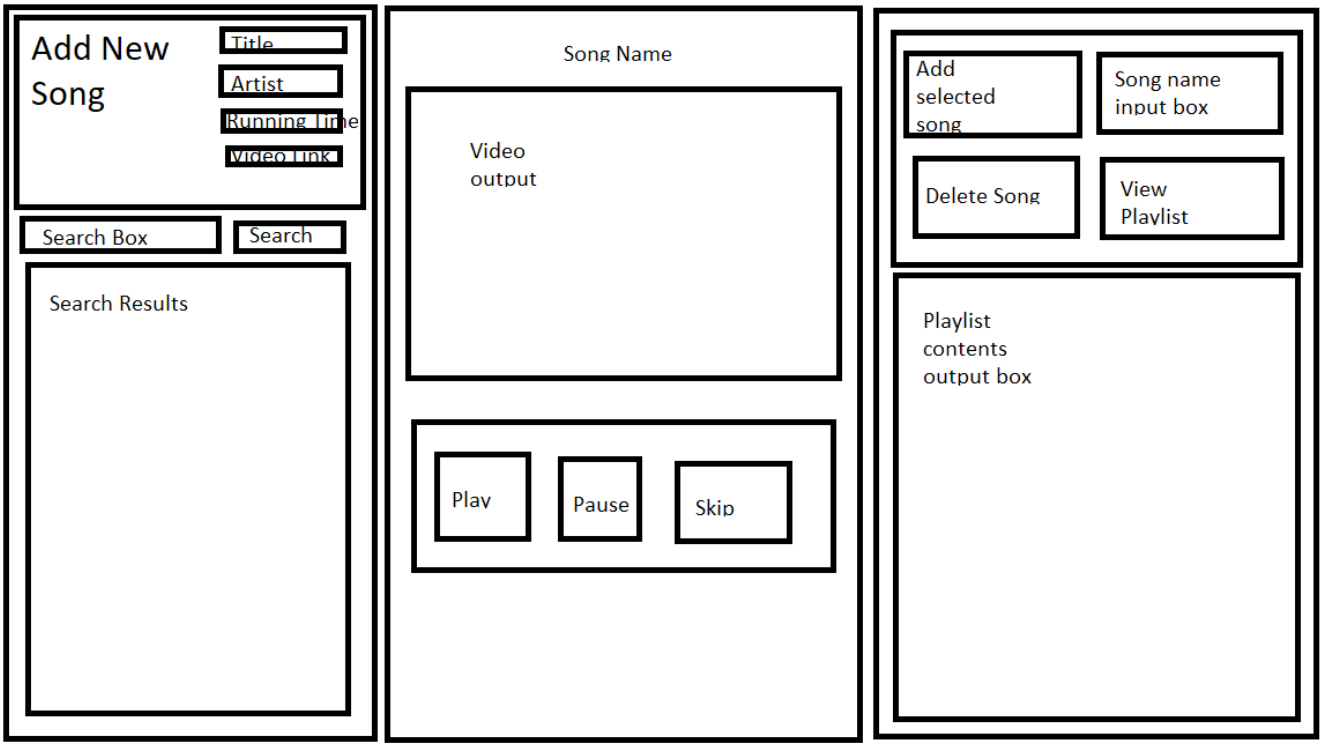
I have managed to produce a karaoke machine that meets the specifications of the project. I am not sure what to say about results and conclusion as this is not a science experiment.

2 Introduction

In this paper I will explain my thought processes and plans for the creation of the Karaoke Machine coursework. I will also discuss and evaluate my success with regards to both the project goals and my own personal goals.

With this report I intend to follow the guidelines set by the project specification on Unihub as best I can. First, I will discuss my plans for the code of the Karaoke machine, explaining the logic for each function the code must perform. I will also show my idea for the GUI for the project. Next, I will showcase the tests that I have performed on the code produced by the above plans. Then I will round off the report with a conclusion reviewing the project. Finally, I will list my references and Java classes in the appendix.

3 Design



Basic GUI wireframe drawing

The program needs to be able to perform 8 different actions:

- Search library for a song
- Add a song to the library by name
- Play the first song in the playlist (and therefore remove it from the playlist)
- Add a song to the playlist by name
- View the contents of the playlist
- Delete any song in the playlist
- Pause the song video playback
- Skip to the next song

It also needs to be able to import songs from any one text file, but this action will be performed upon load and not bound to a button.

I will now explain my plan for how I will code up these actions with psuedocode and measure their time complexity.

3.1 Search library for a song

All songs in the library are in a *HashMap* where the key is the song name so it is possible to search the library *HashMap* for any song by name at hopefully O(1) complexity.

```
function searchLibrary(string songName)
     String output = library.get(songName); C1
     libraryResults.setText(output) C2

T(n) = C1(1) + C2(1)
T(n) = 1
```

This algorithm has time complexity in the order of O(1).

3.2 Add a song to the library by name

Since we access values in the library by song name, adding a song is simple as it does not involve sorting, iterating or shuffling the contents of the library. I just need to create a function that takes all attributes of a song as input, creates a *Song* class object and puts it into the *HashMap*.

```
Function addSong(string songName, string artist, int runningTime,
string fileName)
     Song newSong = new Song(songName, artist, runningTime, fileName)
C1
     library.put(songName, newSong) C2
     return void

T(n) = C1(1) + C2(1)
T(n) = 1
```

This algorithm has time complexity in the order of O(1).

3.3 Play the first song in the playlist (and therefore remove it from the playlist)

I have made the playlist to be a *LinkedList* which has a built in function to access the first item and delete it at the same time. Therefore all the function to play the first song in the playlist needs to do is access the first item via *pollFirst()* then extract the contents of the returned *Song* object and insert it into the GUI. That's the task if no song is playing. If one is active then it needs to unpause the video. Once the first song of a playlist is playing the way to move on is to trigger the skip function either by the button or by having the current song end.

```
Function play()
     if currentSong == null C1
          currentSong = playlist.pollFirst() C2
          titleLabel.setText(currentSong.getTitle) C3
          Media video = new Media(currentSong.getFileName) C4
          // this line is fake, it just shows the concept
          videoView.setMedia(video) C5
     else C6
          // this line is fake, it just shows the concept
          videoView.play() C7

Taking the first option as the worst case:
T(n) = C1(1) + C2(1) + C3(1) + C4(1) + C5(1)
T(n) = 1
```

This algorithm has time complexity in the order of O(1).

3.4 Add a song to the playlist

*LinkedList*s also have a function to add to the end of the list so I can use that to add a new song to the end of the playlist. The function needs to take a song name, search the library for a song of that name and store the resulting *Song* object in the playlist.

```
function addToPlaylist(string songName)
     Song theSong = searchLibrary(songName) C1
     playlist.add(theSong) C2

T(n) = C1(1) + C2(1) // C2 is not C2(n) as the add function can only
add to the end of the linked list so there is no iterating through
the linked list to find the right place each time

T(n) = 1
```

This algorithm has time complexity in the order of O(1).

3.5 View the contents of the playlist

Since the time complexity of this function is not too vital, I think this is best achieved by converting the playlist to an array, then iterating through the array and printing each item’s name to the GUI.

```
function viewPlaylist()
     playlistArray = playlist.toArray() C1
     outputString = “” C2
     for item in playlistArray C3
          outputString = outputString + item.getTitle() + “\n” C4

     playlistContents.setText(outputString) C5

T(n) = C1(n) + C2(1) + C3(n+1) + C4(n) + C5(1)
T(n) = C1(n) + C2 + C3(n+1) + C4(n) + C5
```

This algorithm has time complexity in the order of O(n).

3.6 Delete any song in the playlist

This function also is unlikely to be used as users are more likely to skip over each song as it arrives so the time complexity of this one does not have to be a focus of the code. For this reason, I can just create a blank *LinkedList* then iterate over the current playlist. If the songs name is the requested name then I just skip over it otherwise I add it to the new blank *LinkedList*. Finally I can reassign the playlist variable to the new *LinkedList*. I can also add a call to *viewPlaylist()* to this so the change is immediately obvious.

```
Function deleteFromPlaylist(string songName)
     LinkedList<Song> newPlaylist = new LinkedList<>() C1
     for Song in playlist C2
          if Song.getTitle == songName C3
               // do nothing
          else C4
               newPlaylist.add(Song) C5

     playlist = newPlaylist C6
     viewPlaylist() C7

T(n) = C1(1) + C2(n+1) + C3(n) + C4(n) + C5(n-1) + C6(1) + C7(n)
T(n) = 1 + C2(n+1) + C3(n) + C4(n) + C5(n-1) + C6 + C7(n)
```

This algorithm has time complexity in the order of O(n).

3.7 Pause the song video playback

This function will just be a call to the default function for pausing a video on Java

I will mark it as having time complexity O(1) as it will only be passed one video at a time

3.8 Skip to the next song

This function needs to *pollFirst()* the playlist then copy the returned *Song* object to *currentSong* then insert the *currentSong*'s title and video into the GUI then play the video (if the Java implementation of videos does not already auto play videos that get added).

```
function skipSong()
     currentSong = playlist.pollFirst() C1

     titleLabel.setText(currentSong.getTitle) C2
     Media video = new Media(currentSong.getFileName) C3

     // the next two lines are fake, they just show the concept
     videoView.setMedia(video) C4
     videoView.play() C5

T(n) = C1(1) + C2(1) + C3(1) + C4(1) + C5(1)
T(n) = 1
```

This algorithm has time complexity in the order of O(1)

4. Testing

While I remember being taught about Junit, I have made a conscious decision not to use it as most of my functions are actually procedures, meaning that they have no return value for Junit to check and several have no parameters so that there is no variation in their actions and all edge cases can easily be checked for. I will still take a methodical approach to coming up with test cases to ensure my code works as it should anyway.4.1 Result Tables

4.1.1 ImportMultipleSongs

Expected Input	Given Input	Expected Output	Given Output	Passed
Relative file path to a text file from root directory of Java project	"samplesongdata.txt"	Library *HashMap* filled with *Song* objects and "library imported" printed to console	Library *HashMap* filled with *Song* objects and "library imported" printed to console	Yes
	(no arguments given)	Error caught and program stops	*java.lang.ArrayIndexOutOfBoundsException* raised	Yes (previously no)
	"fakedata.txt" (invalid string)	*FileNotFound Exception* raised	*FileNotFound Exception* raised	Yes

4.1.2 addSong

Expected Input	Given Input	Expected Output	Given Output	Passed
String, *String*, *int*, *String*	*String*, *String*, *String*, *String*	Code will autoconvert *String* to *int* and produce a *Song* object	*NumberFormatException* raised	Yes (previously no)
	One blank box	Code will take blank box as empty string and produce a *Song* object	Code will take blank box as empty string and produce a *Song* object	Yes
	All blank boxes	Code will take blank boxes as empty strings and produce a *Song* object	*NumberFormatException* raised	Yes (previously no)

4.1.3 searchLibrary

Expected Input	Given Input	Expected Output	Given Output	Passed
A *String*	A *String* that matches a song name	Function prints to screen the name of the song and returns the respective *Song* object	Function prints to screen the name of the song and returns the respective *Song* object	Yes
	Random characters	Functions prints “No song with that name found” to the screen and returns null	Functions prints “No song with that name found” to the screen and returns null	Yes
	“” (empty string	Functions prints “No song with that name found” to the screen and returns null	Functions prints “No song with that name found” to the screen and returns null	Yes

4.1.4 addToPlaylist

Expected Input	Given Input	Expected Output	Given Output	Passed
A *String*	A *String* that matches a song name	Function prints to screen the name and artist of the song and adds the song to the playlist	Function prints to screen the name and artist of the song and adds the song to the playlist	Yes
	Random characters	Function tries to search playlist for the string and fails to find	*NullPointerException* raised	Yes (previously no)
	“” (empty string)	Function tries to search playlist for the string and fails to find	*NullPointerException* raised	Yes (previously no)

4.1.5 viewPlaylist

This function has only one test case since you click the button and it either works or does not work.

Expected Input	Given Input	Expected Output	Given Output	Passed
Click button	Click button	Playlist contents are printed to screen, or if playlist is empty, “No songs in playlist is printed instead”	Playlist contents are printed to screen, or if playlist is empty, “No songs in playlist is printed instead”	Yes

4.1.6 deleteFromPlaylist

Expected Input	Given Input	Expected Output	Given Output	Passed
Integer value of a playlist index to delete at	Valid index	Song removed from playlist and “Song removed” printed to console	Song removed from playlist and “Song removed” printed to console	Yes
	Index larger than size of playlist	“No song exists at that index” printed to console	“No song exists at that index” printed to console	Yes
	Negative integer	“No song exists at that index” printed to console	“No song exists at that index” printed to console	Yes
	Non-numerical string	Error raised and caught	*NumberFormatException* raised	Yes (previously no)

4.1.7 playSong

Prerequisite	Expected Output	Given Output	Passed
No song active	Plays first song in playlist	Plays first song in playlist	Yes
Song playing	Prints “Video is already playing” to console	Prints “Video is already playing” to console	Yes
Song paused	Plays video	Plays video	Yes

4.1.8 pauseSong

Prerequisite	Expected Output	Given Output	Passed
No song active	Prints “No video	Prints “No video	Yes

	found” to console	found” to console	
Song playing	Pauses video	Pauses video	Yes
Song paused	Prints “Video was not playing” to console	Prints “Video was not playing” to console	Yes

4.1.9 skipSong

Prerequisite	Expected Output	Given Output	Passed
There is a next song in the playlist	GUI switches to new video and title changes to match	GUI switches to new video and title changes to match	Yes
There is no song in the playlist	Prints “Playlist is empty, cant find next song” to console	Prints “Playlist is empty, cant find next song” to console	Yes

4.1.10 convertCurrentSongtoMediaPlayer

Given Input	Expected Output	Given Output	Passed
CurrentSong = null	Error is caught (This is unlikely to happen as the only time I run this function, I set the value of *currentSong* beforehand.)	GUI reads “There is no available song to play”	Yes (previously no)
CurrentSong = Song object	Function returns a *MediaPlayer* object that plays the video of the filename linked in *currentSong*	Function returns a *MediaPlayer* object that plays the video of the filename linked in *currentSong*	Yes

5. Conclusion

5.1 Summary

I have created a Java program that works as a rudimentary karaoke player, allowing for all functions requested by the project specifications.

- Importing songs to the library

- Searching the library for a song by name
- Adding a song to the playlist by name
- Deleting a song from the playlist
- Viewing the contents of the playlist
- Playing the songs in the playlist in order
- Pausing playback of songs
- Skipping songs in the playlist

5.2 Limitations

My approach has several limitations:

- It can only import songs via text file in the given format or by manual data entry on the user interface. Songs added via the user interface form never enter a text file and only ever exist in the library HashMap and so are deleted upon closing the program. Also, if there is no video file in the location(s) specified by the text file then the program fails.
- As the Song class is currently written, each Song by name can have only one artist and one video. It is possible to enter different versions of the same song into the system each with a different video/artist but…
- Each Song must have a unique name
- If a playlist of n songs is created then played, the code will autoplay the next song as each ends (in addition to skipping on command) but once the last video has finished, if another song is added to the playlist, the code will not autoplay it and the user must press the Skip button to move on to it.
- The space on the user interface only has room to show the first 17 items in the playlist. The playlist itself can hold many more than that but those will not be displayed until the first few are removed
- I don’t know if this counts but the playlist delete function takes indexes as if the data structure starts indexing from one even though it actually starts from zero for ease of use for non-coders.
- You cannot control the volume of the videos except by your devices built in volume controls
- You cannot skip to the beginning of the current video or move the play cursor to like skip the middle or something.
- You cannot rearrange the contents of the playlist without deleting and re-adding the songs until their order is fixed.

There have also been minor changes to most functions to remove redundant code and include code to catch edge cases that makes the current code not match the original plan or tests exactly.

5.3 Future Approach Changes

I think in the future I have no need to make any changes unless required by the project specification. I could probably fix some of the limitations listed above but since some of these either weren’t asked for or aren’t worth marks I don’t see a point in adding them, much as they would extend the project.

6. References

As far as I can tell, this report is completely my work and no part of it is referenced from another persons work.

7. Appendices

The only class created for this project beyond the main program (which is treated as a functional program) and the classes imported and used by Java FX is the class below, called Song.java

```
public class Song implements Comparable<Song>{
    private String title;
    private String artist;
    private int runningTime;
    private String fileName;

    public Song(String title, String artist, int runningTime, String
fileName){
        this.title = title;
        this.artist = artist;
        this.runningTime = runningTime;
        this.fileName = fileName;
    }

    public String getTitle(){
        return this.title;
    }

    public String getArtist(){
        return this.artist;
    }

    public int getRunningTime(){
        return this.runningTime;
    }

    public String getFileName(){
        return this.fileName;
    }

    @Override
    public String toString(){
```

```
        return "\nTitle: " + title + "\nArtist: " + artist + "\
nRunning time: " + runningTime + "\nFile name: " + fileName;
    }

    @Override
    public int compareTo(Song other){
        return getTitle().compareTo(other.getTitle());
    }
}
```